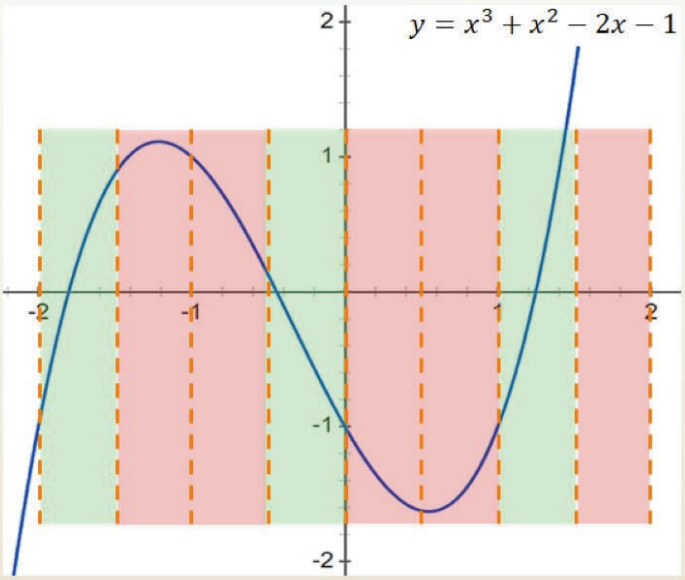
ROOTS: BRACKETING METHODS

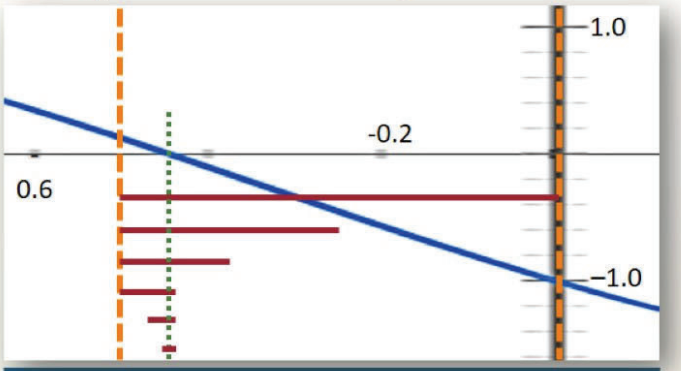
Textbook (Python) Part II Chapter 5

Finding roots of an equation

First, find subintervals containing roots



Then, find root inside a given subinterval

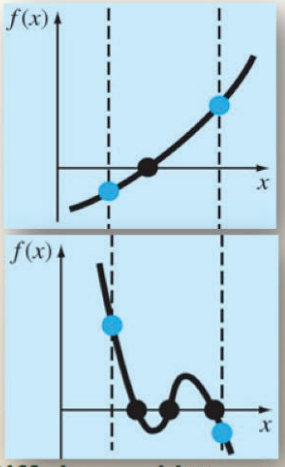


iter	x_l	x_u	x_r
1	**−0.5**	**0**	−0.25
2	−0.5	−0.25	−0.375
3	−0.5	−0.375	−0.4375
4	−0.5	−0.4375	−0.4688
5	−0.4688	−0.4375	−0.4531

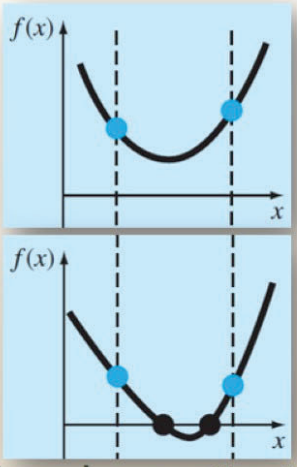
The Incremental search method

Input: a function f and an interval i
Output: subintervals of i containing a root

Evenly divide i into some n subintervals, then for each subinterval do $f(x_l) \times f(x_u) < 0$



Diff signs, odd # roots

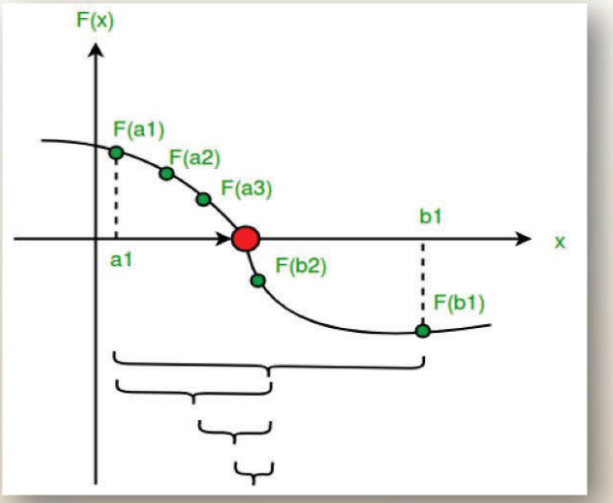


Same sign, even # roots

The bisection method

Input: f and an interval that contains a root
Output: an approximation of the root of f, i.e. the value of x such that $f(x) = 0$

Iteratively halve the search region bracketed the root



https://www.geeksforgeeks.org/program-for-bisection-method/

Incremental Search

- Find intervals that contain the roots
- *The source code is from the textbook, figure5.4*

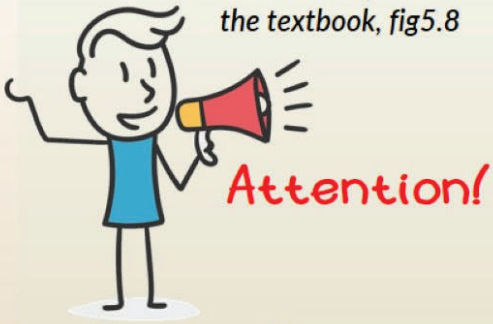
Attention!
This updated version is uploaded to Wk04

```
import numpy as np
def incsearch(func,xmin,xmax,ns=50):
""" incsearch: incremental search locator
    incsearch(func,xmin,xmax,ns) find brackets of x that contain sign
changes in a function of x on an interval
  input:
    fun = name of the function
    xmin, xmax = endpoints of the interval
    ns = number of subintervals, default value = 50
  output:
    nb = number of bracket pairs found
    xb = list of bracket pair values
    or returns "no brackets found" """

  x = np.linspace(xmin,xmax,ns+1) # create array of x values
  f = [] # build array of corresponding function values
  for k in range(ns+1):
    f.append(func(x[k]))

  nb = 0
  xb = []
  for k in range(ns): # check adjacent pairs of function values
      if (func(x[k]) * func(x[k+1]) < 0): # for sign change
        nb = nb + 1 # increment the bracket counter
        xb.append((x[k],x[k+1])) # save the bracketing pair
  if nb==0:
    return 'no brackets found'
  else:
    return nb, xb
```

```
incsearch(lambda x: np.sin(10*x) + np.cos(3*x),3,6)
```

Bisection

- Notice how ε_a and its absolute is computed
- Both ε_a and ε_s in the code is not a percentage
  - No deduction for HW03, but be sure to do it correctly from now onwards
- *This code is from the textbook, fig5.8*

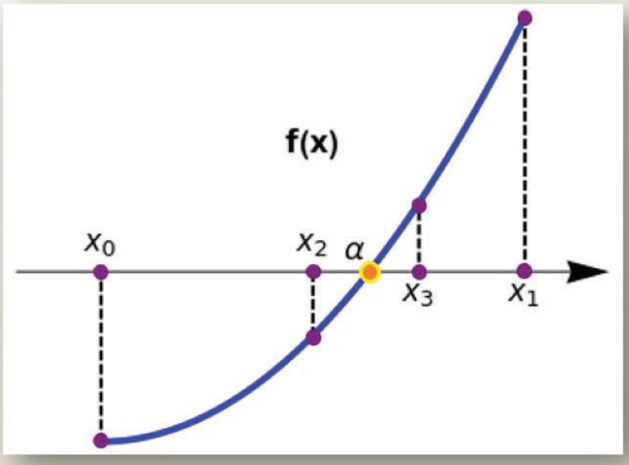


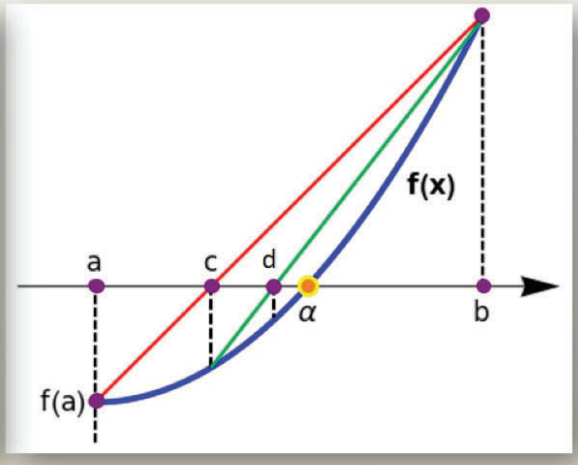

```
def bisect(func,xl,xu,es=1.e-7,maxit=30):
""" Uses the bisection method to estimate a root of func(x). The method is
iterated until the relative error from one iteration to the next falls below the
specified value or until the maximum number of iterations is reached first.
    Input:
        func = name of the function
        xl = lower guess
        xu = upper guess
        es = relative error specification (default 1.e-7)
        maxit = maximum number of iterations allowed (default 30)
    Output:
        xm = root estimate
        fm = function value at the root estimate
        ea = actual relative error achieved
        i+1 = number of iterations required or error message if initial guesses
do not bracket solution """
    if func(xl)*func(xu)>0:
        return 'initial estimates do not bracket solution'
    xmold = xl
    for i in range(maxit):
        xm = (xl+xu)/2
        ea = abs((xm-xmold)/xm)
        if ea < es: break
        if func(xm)*func(xl)>0:
            xl = xm
        else:
            xu = xm
        xmold = xm
    return xm,func(xm),ea,i+1
```

```
bisect(lambda x: np.sin(10*x) + np.cos(3*x),xl,xu)
```


Bisection vs. False position

- Each iteration
  - Approximate the root and compute ε_a
  - The search interval is divided into two sections
  - Keep the section that contains the root $f(x_l) \times f(x_u) < 0$



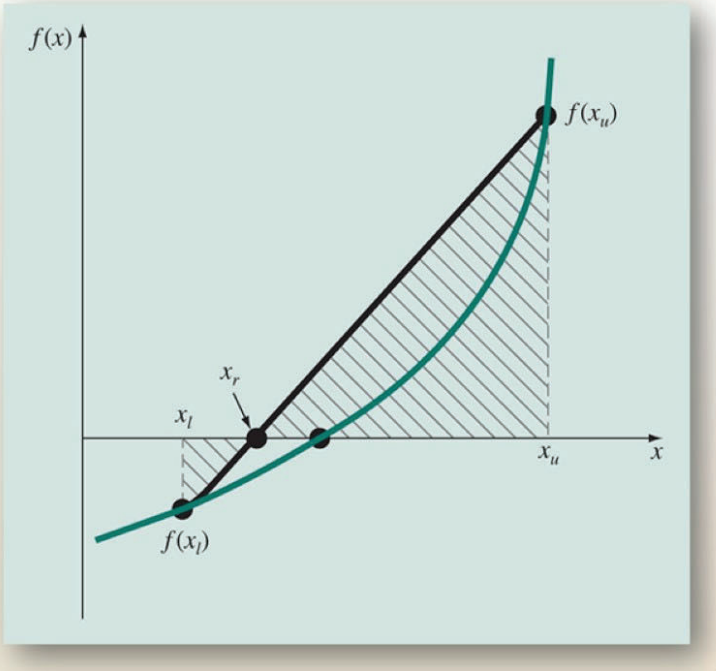


https://www.cfm.brown.edu/people/dobrush/am33/Mathematica/ch3/bracket.html

The false position method

- Another bracketing method
- The next guess, that is x_r, is obtained not by splitting the bracketing interval in half but by
  - connecting the endpoints, $f(x_l)$ and $f(x_u)$, with a straight line
  - then determining the location where the line crosses the x-axis

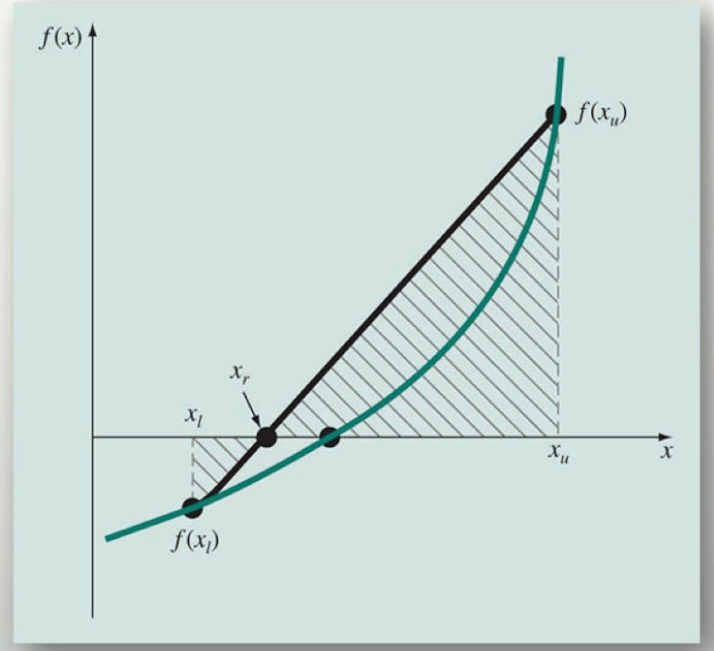
$$x_r = \frac{f(x_u)\cdot x_l - f(x_l)\cdot x_u}{f(x_u) - f(x_l)}$$

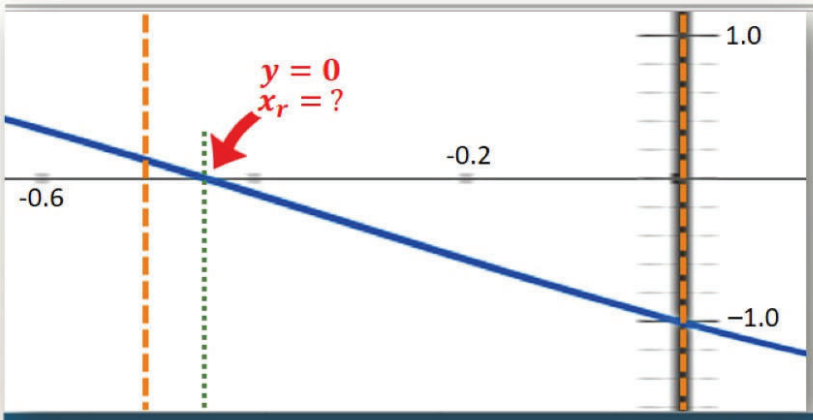


PRACTICE PROBLEMS

- Derive the formula for x_r

$$x_r = \frac{f(x_u)\cdot x_l - f(x_l)\cdot x_u}{f(x_u) - f(x_l)}$$

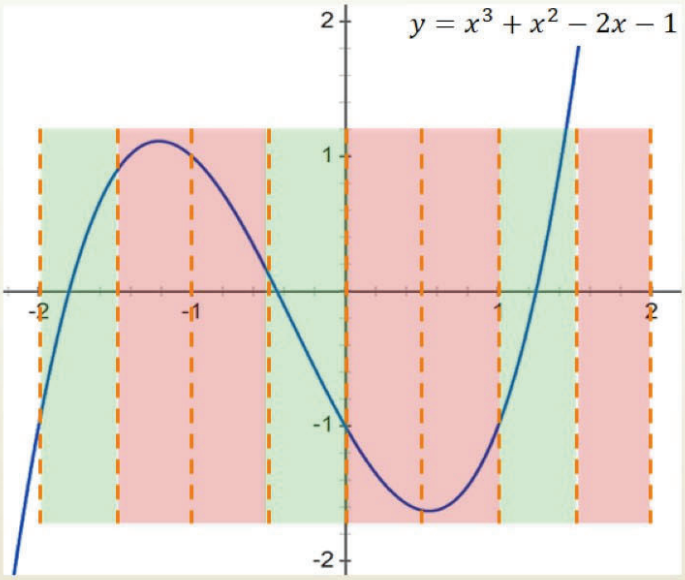




- Use false position to find the root of $y = x^3 + x^2 - 2x - 1$
- Assume we have already located a root inside the interval $[-0.5, 0]$
- For results to be correct to at least 3 sig figs, i.e. $\varepsilon_s = 0.05\%$, the number of iterations = ___

iter	x_l	x_u	x_r	$f(x_l)$	$f(x_u)$	$f(x_r)$	ε_a
1	-0.5	0					
2							
3							
4							
5							

Bisection vs. False position



Bisection

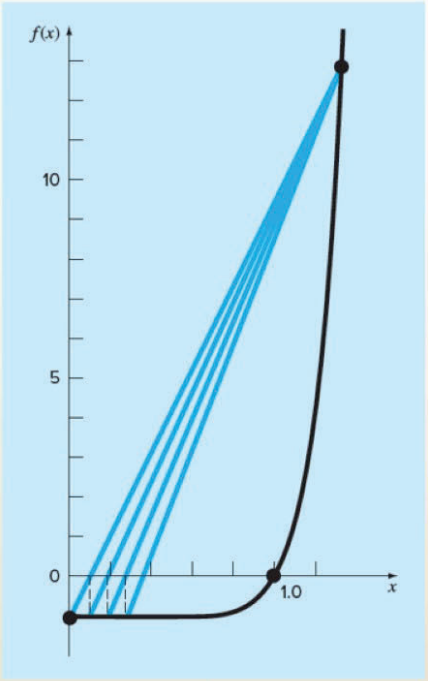
root	# iterations	x_r	ε_t
A	10	-1.8022	-0.0308%
B	12	-0.4449	0.0096%
C	10	1.2466	-0.0398%

False position

root	# iterations	x_r	ε_t
A	5	-1.8019	-0.0078%
B	3	-0.4450	-0.0000%
C	5	1.2469	-0.0088%

When bisection is preferred over false position

- False position is based on the idea that if $f(x_l)$ is much closer to zero than $f(x_u)$, it is likely that the root is also closer to x_l than x_u.
- When such condition is violated, then false position may not be a preferred method.
- E.g. on the right is a graph of $f(x) = x^{10} - 1$



try IT! Solve this using bisection and false position and compare the results. Find the root in the interval [0,1.3], run it for 5 iterations

ROOTS: OPEN METHODS

Textbook (Python) Part II Chapter 6

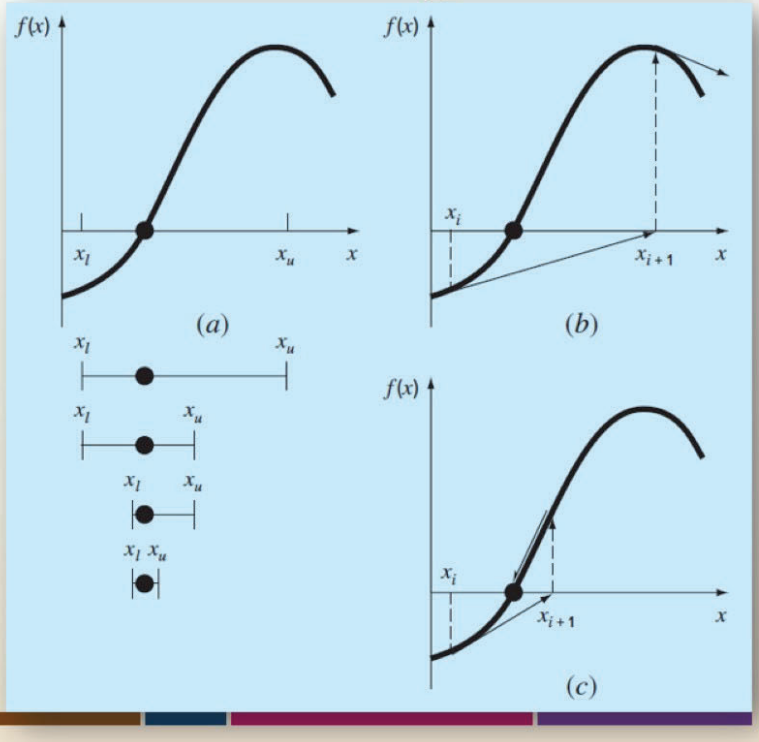
Root finding methods: open vs bracketing

- Open methods are based on formulas that require only a single starting value of x or two starting values that not necessarily bracket the root

(a) Bracketing method

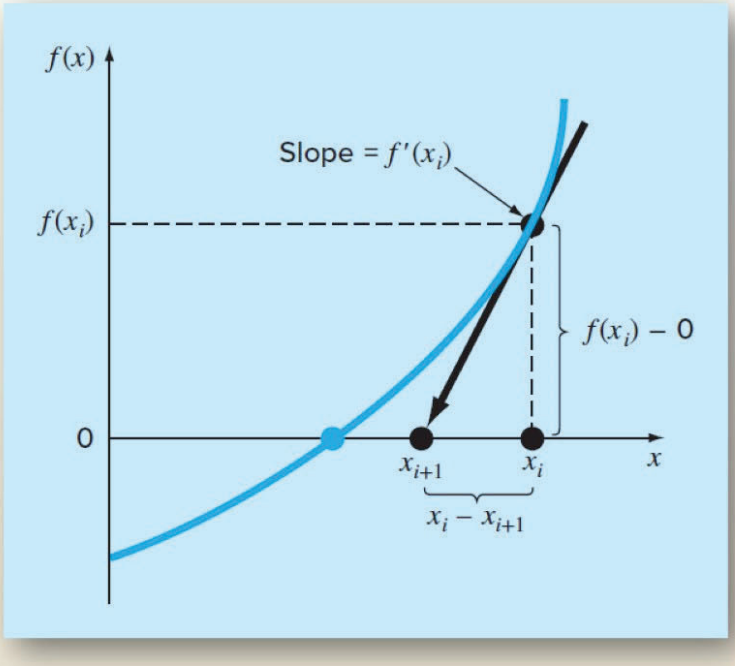
(b) Diverging open method

(c) Converging open method
notice the speed!

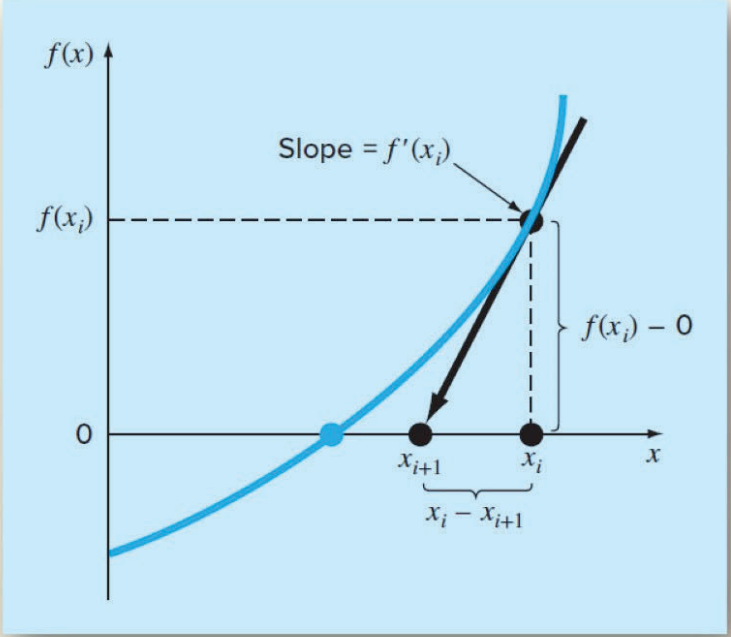


Graphical depiction of the Newton-Raphson method

- Forming the tangent line to the $f(x)$ curve at some guess x, then following the tangent line to where it crosses the x-axis.
- Convenient for functions whose derivatives can be evaluated analytically



Graphical depiction of the Newton-Raphson method



WORKED EXAMPLES

- Consider the function $f(x) = x^2 - x - 1$. Solve it analytically. Then solve it using the open method, and compare the results.

WORKED EXAMPLES

- Consider the function $f(x) = x^2 - x - 1$. Solve it analytically. Then solve using Newton-Raphson method with an initial guess of $x = 5$. Take the approximate root in the second iteration and compute ε_t.

Iter	Approximate root	ε_t (%)	$f(x_i)$	$f'(x_i)$
0	5			
1				
2				

WORKED EXAMPLES

- Consider the function $f(x) = x^2 - x - 1$. Solve it analytically. Then solve using Newton-Raphson method. Determine one possible initial guess that would lead the algorithm to return the negative root.

Iter	Approximate root	ε_t (%)	$f(x_i)$	$f'(x_i)$
0				
1				
2				

PRACTICE PROBLEMS

- Use the Newton-Raphson method to estimate the root of $f(x) = x - e^{-x}$ employing an initial guess of $x_0 = 0$.

True value: 0.567143140453502

Iter	Root	ε_t (%)	$f(x_i)$	$f'(x_i)$
0	0			
1				
2				
3				

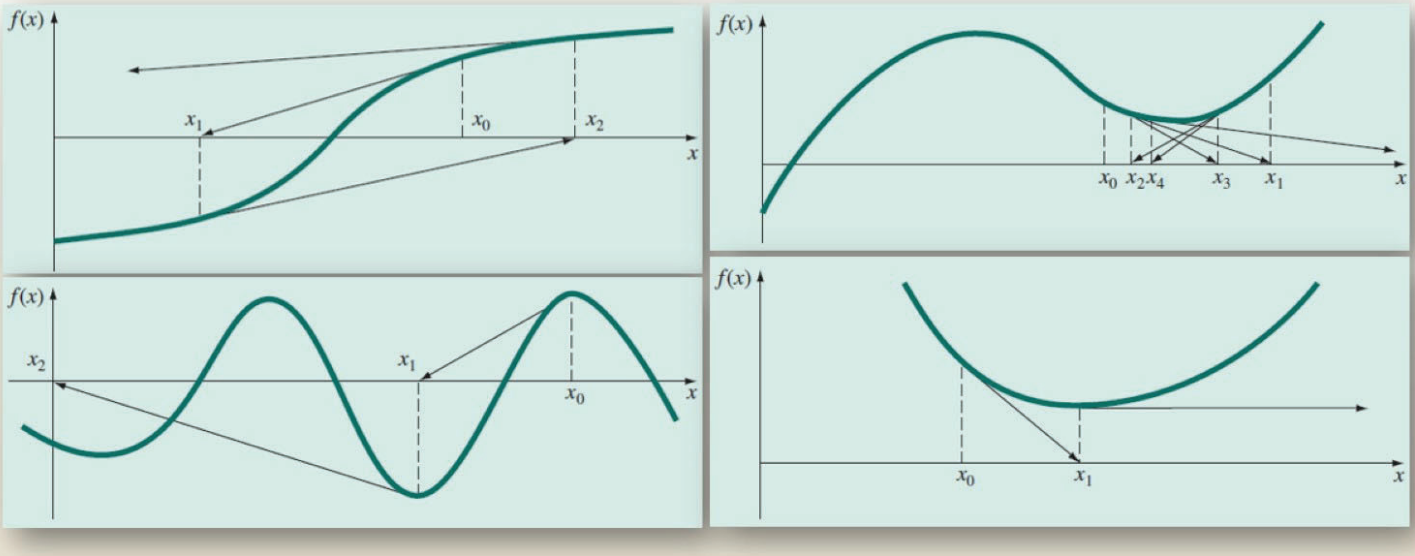
Ex.6.4

PRACTICE PROBLEMS

- Determine the positive root of $f(x) = x^{10} - 1$ using the Newton-Raphson method and an initial guess of $x = 0.5$

Cases when Newton-Raphson is poor convergence

- Some functions show slow or poor convergence or divergence!



Newton-Raphson

- An open method for locating the root in a given search interval
- *The source code is from the textbook, figure6.11*

```
def newtraph(f,fp,x0,Ea=1.e-7,maxit=30):
  """
  This function solves f(x)=0 using the Newton-Raphson method.
  The method is repeated until either the relative error
  falls below Ea (default 1.e-7) or reaches maxit (default 30).
  Input:
    f = name of the function for f(x)
    fp = name of the function for f'(x)
    x0 = initial guess for x
    Ea = relative error threshold
    maxit = maximum number of iterations
  Output:
    x1 = solution estimate
    f(x1) = equation error at solution estimate
    ea = relative error
    i+1 = number of iterations
  """
  for i in range(maxit):
    x1 = x0 -f(x0)/fp(x0)
    ea = abs((x1-x0)/x1)
    if ea < Ea: break
    x0 = x1
  return x1,f(x1),ea,i+1
```

```
(xsoln,ysoln,ea,n)=newtraph(lambda x:x**10-1,lambda x:10*x**9,x0)
```